006570.P033 *PATENT*

UNITED STATES PATENT APPLICATION

FOR

UNIFIED LOGGING SERVICE WITH A LOGGING FORMATTER

INVENTOR:

GREGOR K. FREY
HEIKO KIESSLING
MIROSLAV R. PETROV
GEORGI G. MANEV
NIKOLA I. MARCHEV

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN, LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(503) 684-6200

EXPRESS MAIL NO.

EV325529582US

UNIFIED LOGGING SERVICE WITH A LOGGING FORMATTER

TECHNICAL FIELD

[0001] Embodiments of the invention generally relate to the field of data processing systems and, more particularly, to a system and method for a unified logging service having a logging formatter.

BACKGROUND

[0002] Logging is employed within virtually all data networks. "Logging" refers generally to recording network-related and/or application-related information in response to one or more predefined network/application events. For example, when an end-user opens a TCP connection to a server, or unsuccessfully attempts to gain access to network resources (e.g., by attempting to log in to a particular server), this information is typically recorded as an entry within a log file. Similarly, if a variable within an application rises above a specified threshold value, a log entry indicating the value and the date and time that the threshold value was exceeded may be stored within a log file. Logging techniques may be employed to record any specified network/application event. Network administrators may then review the log files to identify security issues and/or troubleshoot network problems.

[0003] Logging functionality is provided within the Java™ 2 Standard Edition ("J2SE™") platform and the Java 2 Enterprise Edition "J2EE™" platform. Referring to FIG. 1, in a Java environment, Java applications 101 make logging calls on "logger" objects 110, 112, 114. Each logger object is used to log messages for a specific system or application component. Loggers are normally named using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they are typically based

on the package name or class name of the logged component (e.g., such as java.net or javax.swing). In addition, it is possible to create "anonymous" loggers that are not stored in the logger namespace. Loggers are organized in a hierarchical namespace in which child loggers 112, 114 may inherit logging properties from their parents 110 in the namespace.

[0004] Each logger 110, 112, 114 may have a threshold "Level" associated with it which reflects a minimum defined logging value (e.g., priority level) that the logger cares about. If a logger's level is set to null, then its effective level is inherited from its parent, which may in turn obtain it recursively from its parent, and so on up the tree.

[0005] In response to logging calls from applications 101, the logger objects 110, 112, 114 allocate Log Record objects which are passed to handler objects 130 for publication. For example, a first type of handler object may write log records to an output stream, a second type of handler object may write log records to a file (or to a set of rotating log files) and a third handler may write log records to remote TCP ports. Developers requiring specific functionality may develop a handler from scratch or subclass one of the handlers in J2SE.

[0006] Both loggers 110, 112, 114 and handlers 130 may use filters 120, 121 to filter out certain designated types of log records. In addition, when publishing a log record externally, a handler may optionally use a formatter 122 to localize and format the message before writing it to a particular destination. For example, J2SE includes a "simple formatter" for writing short "human-readable" summaries of log records and an eXtensible Markup Language (XML) formatter for writing detailed XML-structured information.

[0007] "Tracing" is a technique used primarily by software developers to track the execution of program code. For example, when developing an application, developers trace the execution of methods or functions within certain modules to identify problems and/or to determine if the program code may be improved. If a particular method takes an inordinate amount of time to complete, the developer may determine the reasons why and/or change the program code to operate more efficiently.

[0008] Developers use trace tools to trace the execution of program code. Trace tools are proprietary application programs which use different techniques to trace the execution flows for an executing program. One technique, referred to as event-based profiling, tracks particular sequences of instructions by recording application-generated events as they occur. By way of example, a trace tool may record each entry into, and each exit from, a module, subroutine, function, method, or system component within a trace file (e.g., a time-stamped entry may be recorded within the trace file for each such event). Trace events may also be sent to a console or other output destination.

[0009] Thus, tracing and logging techniques rely on similar event-based triggers, employ similar messaging techniques and record log/trace events to similar output destinations (e.g., trace/log files, consoles, . . ., etc.) in a substantially similar manner. As such, it would be beneficial to develop an integrated application programming interface which takes advantage of the similarities of tracing and logging operations, and of the synergistic effects of handling both, while not neglecting the differences.

SUMMARY OF THE INVENTION

[00010] An integrated tracing and logging system for an enterprise network is described. One embodiment of the integrated logging and tracing system has an object-oriented architecture which includes a controller class with two sub-classes: a tracing sub-class and a logging sub-class. Instances of the tracing sub-class (tracing modules) are associated with specified program code regions of applications. The tracing modules receive method calls from the applications and process the method calls based on defined severity levels. Instances of the logging sub-class (logging modules) are associated with specified "categories" related to the enterprise network (e.g., system, database, etc.). The logging modules receive and process method calls from network components associated with the categories. The integrated logging and tracing system allows tracing and logging information to be collected and correlated in a variety of useful ways. In an embodiment, the integrated logging and tracing system may use a configurable formatter to provide a message format for logging/tracing messages.

BRIEF DESCRIPTION OF THE DRAWINGS

[00011] Embodiments of the invention are illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like reference numerals refer to similar elements.

Figure 1 illustrates a prior art system for performing logging operations.

Figure 2 illustrates a system for performing both logging and tracing operations according to one embodiment of the invention.

Figure 3 illustrates hierarchical relationships among tracing and logging controllers according to one embodiment of the invention.

Figure 4 illustrates one embodiment of the invention in which severity settings of trace/log controllers are configured using inheritance.

Figure 5 illustrates a method for configuring logging and tracing according to one embodiment of the invention.

Figures 6a-b illustrate severity levels associated with tracing and logging controllers according to one embodiment of the invention.

Figure 7a illustrates selected elements of an exemplary configuration file 700, according to an embodiment of the invention.

Figure 7b illustrates a default log/trace message format as defined by configuration file 700.

Figure 8 illustrates selected elements of an exemplary configuration file 800, according to an embodiment of the invention in which the message format is configurable.

Figure 9 is a flow diagram illustrating certain aspects of a method for configuring the message format of a log/trace message, according to an embodiment of the invention.

Figure 10 is a block diagram of computing device 1000 implemented according to an embodiment of the invention.

DETAILED DESCRIPTION

[00012] Embodiments of the invention are generally directed to an integrated logging and tracing system having a formatter to provide a message format for logging and tracing messages. One embodiment of the integrated logging and tracing system has an object-oriented architecture which includes a controller class with two sub-classes: a tracing sub-class and a logging sub-class. Instances of the tracing sub-class (tracing modules) are associated with specified program code regions of applications. The tracing modules receive method calls from the applications and process the method calls based on defined severity levels. Instances of the logging sub-class (logging modules) are associated with specified "categories" related to the enterprise network (e.g., system, database, etc). The logging modules receive and process method calls from network components associated with the categories. A formatter may provide a message format for logging and/or tracing messages sent from the logging and/or tracing modules. As is further described below, in an embodiment, the message format may be configured without recompiling any source code.

[00013] A system architecture according to one embodiment of the invention is illustrated in FIG. 2. The architecture includes a plurality of controllers 200 managed by a log/trace manager 210. The controllers 200 are configured to process trace/logging events generated by a plurality of different applications 201. As used herein, the term "application" is used broadly to refer to any type of program code executed on a computer and/or transmitted over a network (e.g., using a particular network protocol). One embodiment of the invention is implemented in an object-oriented programming

environment such as Java (e.g., within a Java 2, Enterprise Edition (J2EE) platform/engine). In this embodiment, each of the modules illustrated in FIG. 2 are represented by objects and/or classes. The classes and/or objects of this embodiment comprise an Application Programming Interface ("API") usable to configure logging and/or tracing operations within a Java environment. It should be noted however, that the underlying principles of the invention are not limited to any particular programming environment.

[00014] In one embodiment, each controller 200 is an instance of a defined "controller" class (e.g., a Java class) which includes two sub-classes, a "tracing" sub-class and a "logging" sub-class (described in detail below), which provide features specific to tracing and logging operations, respectively. In an object-oriented environment such as Java, the tracing controller 202 illustrated in FIG. 2 represents an instance of the tracing sub-class and the logging controller 204 represents an instance of the logging sub-class. In one embodiment of the invention, tracing controllers 202 are associated with program code locations (e.g., locations within packages, classes, . . ., etc.) whereas logging controllers 204 are associated with system categories (e.g., logical system categories such as database, network, . . ., etc.).

THE CONTROLLER CLASS

[00015] The controller class provides methods for associating log/trace output destinations with specific controllers 200 and for controlling the actual writing of log/trace messages. When a method is called, the writing of the log/trace message to a log/trace file 220, console 221 or other output destination 214 depends on the severity

level associated with the message, the severity settings 205, 206 of the relevant controller(s) 200, and the filtering configuration of one or more optional filters 212, 216. For example, in one embodiment, messages having a severity level greater than or equal to the effective severity of the relevant controller 200 are candidates for output and are forwarded to the output destinations 214 attached to the controller 200 (e.g., assuming that the messages are not filtered by one of the filters 212, 216).

[00016] A variety of different severity levels may be defined. In one embodiment of the invention, illustrated in FIG. 6a, the following severity levels are defined (from lowest to highest severity level): MINIMUM, DEBUG, PATH, INFO, WARNING, ERROR, FATAL, and MAXIMUM. The general description for each severity level is set forth in FIG. 6a. In addition, as illustrated in FIG. 6b, in one embodiment, logging and tracing may be totally enabled or disabled using the severity settings of ALL or NONE, respectively. As indicated in the second column of FIGs. 6a-b, each of the descriptive threshold levels may have an associated numeric value (e.g., DEBUG = 100, PATH = 200, . . ., etc.).

[00017] Before (or after) evaluating the trace/log message based on severity, filters 212 associated with the controller 200 may filter the messages based on predefined filtering criteria. By way of example, if a particular controller 200 is capable of writing to both a file and a console, a filter 212 may be assigned to filter messages directed to the file, thereby limiting output to the console only. Thus, using filters, a finer granularity of log controller 200 output may be defined, using variables other than severity. As illustrated in FIG. 2, filters may be associated with particular log controllers 200 and/or with specific output destinations 214 (e.g., specific log/trace files). As mentioned above,

filters may be associated with both controllers 200 and/or output destinations 214 to further restrict or alter the output tracing and logging behavior. In one embodiment, multiple filters having different filtering criteria may be allocated to each controller 200 and/or destination 214.

THE LOGGING SUB-CLASS

[00018] As mentioned briefly above, in one embodiment, the logging operations performed by the logging controller 204 are associated with particular "categories" which may identify, for example, semantical topics which correspond roughly to administration tasks or problem areas. Typical areas identified by categories may include databases, networking, security and auditing.

[00019] In one embodiment, categories are named according to the hierarchical structure known from file systems on the network. For example, referring to FIG. 3, all categories of log messages associated with the system 311 may be logically grouped into sub-categories, such as database 312 or networking 313, identified by the names "/System/Database" and "/System/Network," respectively. In this example, the category "/System" 311 is the parent of categories "/System/Database" and "/System/Network," and therefore passes on its settings (e.g., severity thresholds) and attached output destinations 214 to both of them. One benefit to this arrangement is that if all log messages are to be written to the same log file, the log file does not necessary need to be directly associated with both the database 312 and networking 313 categories, but simply to the common parent, System 311.

THE TRACING SUB-CLASS

[00020] In one embodiment, in contrast to logging operations which are associated with categories, tracing operations performed by the tracing controllers 202 are associated with particular program code locations, identified by particular package, class, or, function names. For example, in a Java environment, locations may be named according to the hierarchical structure known from Java packages.

[00021] In the example illustrated in FIG. 3, the location "com.sapmarkets" 301 is logically grouped into technology and application sub-locations, identified by the controller names "com.sapmarkets.technology" 302 and "com.sapmarkets.application" 303, respectively. In this example, the location "com.sapmarkets" 301 is the parent of locations "com.sapmarkets.technology" 302 and "com.sapmarkets.application" 303," and therefore passes on its settings (e.g., severity thresholds) and attached output destinations 214 to both of them. As with the logging controllers, a benefit to this configuration is that if all trace messages are to be written to the same trace file, the trace file does not need to be directly associated with both the technology and application locations, but simply to the common parent, "com.sapmarkets" 301.

[00022] As an additional example, to write all the messages from monitoring classes into a single log, the location named "com.sap.tc.monitoring" may be called. All the messages from technology components (TC) may be directed to a common trace destination by simply assigning that destination to the parent location "com.sap.tc." The trace output destination is then passed on to all child locations (e.g., to "com.sap.tc.monitoring"). Moreover, it is possible to include method signatures in location names. For example, "com.sap.tc.monitoring. Node.announce (java.lang.Object)" is a location for a method named "announce" with an argument of the

class Object. In this way, overloaded methods can be identified and, by adding another suffix to such a name, even classes local to them. In a Java environment, the hierarchical components of the name should be compliant with Java identifier syntax, but illegal characters may be used, bracketing a component with single quotes (e.g., com.sap.'great.technology').

[00023] In one embodiment, all locations are accessed via a defined static method "Location.getLocation." In one embodiment, the static method need not be called each time a trace message is to be generated. Instead, each class defines static fields to hold all the needed locations, and uses these fields to call logging/tracing methods.

CONTROLLING SEVERITY SETTINGS USING INHERITANCE

[00024] In one embodiment, the various controllers 200 are arranged into logical hierarchies in which each child controller inherits properties from its parent (e.g., its severity settings and its output destinations 214). The effective severity of a child log controller is calculated from *minimum* and *maximum* defined severities, as well as the effective severity of its parent. The minimum severity setting specifies the minimum severity value for the trace/log messages to be output via the child log controller. That is, if the effective severity of the parent log controller is higher than the minimum severity of the child, then the child inherits the parent's effective severity; otherwise the effective severity is set to the minimum severity.

[00025] By contrast, the maximum severity represents the severity that trace/log messages must have for output to be guaranteed (notwithstanding intervention from filters). For example, if the effective severity of the parent log controller is lower than

the maximum severity of the child then the child inherits the parent's effective severity; otherwise the child's effective severity is set to the maximum severity. This implies that if both minimum and maximum severity are set to the same value, the effective severity is set to that value regardless of the effective severity of the parent. Such a setting is therefore called *dominant*. For root log/trace controllers the setting must be dominant, as separate minimum and maximum values are meaningless without a parent to inherit from.

[00026] FIG. 4 illustrates an exemplary set of tracing controllers 202 which are configured through inheritance. Specifically, tracing controller "com.sapmarkets," the root controller, is configured with an effective severity of INFO. Controller "com.sapmarkets.technology" 302, a child of controller 202, is programmed with a minimum severity level of WARNING and a maximum severity level of FATAL. Because the effective severity of parent controller 301 is lower than the minimum severity setting of child controller 302, the effective severity of the child controller 302 is its minimum severity setting, WARNING.

[00027] Similarly, tracing controller "com.sapmarkets.technology.monitoring" 401, is programmed with a minimum severity of INFO and a maximum severity of ERROR. Accordingly, because the effective severity of its parent controller 302 (WARNING) is higher than its minimum severity, INFO, and lower than its maximum severity, ERROR, tracing controller 401 inherits its parent's effective severity of WARNING.

[00028] By contrast, tracing controller "com.sapmarkets.application" 303 is manually configured with an effective severity of PATH, which overrides the effective severity of INFO inherited from its parent tracing controller "com.sapmarkets" 301.

[00029] It should be noted, of course, that the specific details set forth above are used merely for the purpose of illustration. The underlying principles of the invention are not limited to any particular logging/tracing controller hierarchy or any particular severity settings.

OUTPUT FORMATTERS

[00030] As illustrated in FIG. 2, one embodiment of the invention includes a plurality of log/trace formatters 218 to format the results of the log/trace controllers 200, for example, based on the specified destination to which the controller output is directed. Three formatters are illustrated in FIG. 2: a list formatter 230, a human-readable formatter 231, and an eXtensible Markup Language (XML) formatter 232.

[00031] In one embodiment, the list formatter 230 translates the output of the log/trace controllers into a format which may be further processed by an application, e.g. a log viewer, instead of being read directly by an end-user. In one embodiment, the format generated by the list formatter comprises hash-separated fields which may be readily interpreted by the other applications. For example:

"#1.3#10.48.27.165:4A5AB2:E99D42D4F4:-8000#Mon Jan 01 22:00:00PDT2001#com.sapmarkets.FooClass#com.sapmarkets.FooClass.fooMethod#main##0#0#Fatal##Plain###A sample fatal message#."

[00032] As its name suggests, the human readable formatter 231 translates the output of the log/trace controllers into a human-readable format (e.g., ASCII text). As such, this formatter may be used when users/administrators need to quickly understand what is occurring within a network or application. For example, the human readable formatter 231 may provide its formatted output to a trace/log console for immediate viewing.

[00033] The markup language formatter 232 translates the output of the log/trace controllers into a particular markup language such as the eXtensible Markup Language ("XML") format which may be interpreted by any application that includes support for XML (e.g., Microsoft Word).

RELATIONSHIP BETWEEN LOGGING AND TRACING OPERATIONS

[00034] Messages associated with a particular *category* may also be associated with (e.g., may be written with respect to) a source code area, or *location*, such as a component, a package, a class or a method. As the location may be associated with a particular tracing controller, the same method call can write a message simultaneously to, for example, the database log as well as to the location trace responsible for that part of the source code (e.g., save for the location having an appropriate severity setting). In one embodiment of the invention, both the trace message and the log message are generated with the same identification in order to facilitate cross-referencing among location and category logs. At the same time, the location provides a location description, that is a string, to the log message written to the database log. This may become tedious when a class implements a large number of methods. Therefore, as described in greater detail

below, for each logging/tracing method there is a version which takes in an additional parameter, referred to herein as "subloc," which is a string that serves as suffix to the name of the used location, thus providing the option to give a complete location name whilst avoiding clutter.

EXAMPLES OF TRACING/LOGGING OPERATION

[00035] For the purpose of illustration, one particular tracing/logging example will now be described with respect to the method outlined in FIG. 5. The example describes how tracing/logging may be enabled and configured within the architecture illustrated in FIGs. 2-4. Briefly, the method is comprised of: identifying the source code area for which trace/log output is desired (501); assigning severity levels to the source code (502); specifying the output destination for the logs/traces (503); and inserting messages with corresponding severity levels (504).

[00036] The following sample code will be used for the example:

```
package com.sap.fooPackage;
import com.sap.tc.logging.*;
 public class Node {
   private static final Location loc =
 Location.getLocation("com.sap.fooPackage.Node");
   public void announce(Object o) {
      String method = "announce(java.lang.Object)";
     loc.entering(method);
     try{
       // do something...
      loc.debugT(method, "Connecting to …");
     }
     catch (Exception e) {
       loc.fatalT(method,
```

```
                    "Error processing object {0}",
                    new Object[] {o});
            }
            loc.exiting();
        }
    }
```

[00037] Method elements 502 and 503 are not shown at this point, but assuming the severity level is set to be *Severity.ALL* (accept all severity levels and output everything) and output has been formatted using a human-readable formatter (e.g., and sent to a console) the output may look like the following:

```
May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main]  Path:
Entering method
May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main]  Debug:
Connecting to ….
 May 3, 2001 6:54:18 PM
 com.sap.fooPackage.Node.announce [main]  Path:
 Exiting method
```

[00038] The following four sections will further explain each step illustrated in FIG. 5.

Identifying the Source Code Area (501)

[00039] As described above, the tracing subclass and the logging subclass are subclasses of the controller class. The tracing subclass is sometimes referred to below as the "*Location*" subclass and the logging subclass is sometimes referred to below as the

"*Category*" subclass. Recall that *Location* is the source area that generates trace messages. Typically, it corresponds to the source code structure, and can be attached to the level of component, package, class or method. Category is the source area that generates log messages, corresponding to a distinguished problem area, such as networking, system and database.

[00040] Although the naming of a *Location* and *Category* may be quite flexible, as a general rule, a valid hierarchical naming convention may be useful. A common naming practice for *Location* is to use the full path of Java package name (See, e.g., FIGs. 3-4). For the purpose of this example, the following static methods are provided for each class for easy access to a *Location* or *Category*:

```
Location.getLocation(<name of the Location>);

Category.getCategory(<name of the Category>).
```

[00041] Alternatively, instead of passing the name manually, for *Location* the class instance may be passed (`java.lang.Object`) or the class itself (`java.lang.Class`). In either case, the *Location* object is by default referring to the class level, while using the string argument (`java.lang.String`) provides flexibility in the definition (e.g., to also include the method name to explicitly control logging over methods individually).

[00042] Once identified, the source is ready to be configured to generate messages. In one embodiment, the handle may be configured to be static to improve efficiency: "`static final Location loc = Location.getLocation (this.getClass()).`"

Assign Severity to Source (502)

[00043] Recall that the severity levels employed in one embodiment of the invention are set forth in FIG. 6a-b. Thus, the following predetermined constants may be used: *Severity.DEBUG, Severity.PATH, Severity.INFO, Severity.WARNING, Severity.ERROR, Severity.FATAL Severity.MAX, Severity.ALL* and *Severity.NONE.* The severity may be assigned to the specified source using the following method: "`loc.setEffectiveSeverity(Severity.INFO);`." Using this configuration, any messages with severity lower than *INFO* will be discarded. Others will be directed to the destination. The concepts associated with hierarchical severity were described above with respect to FIG. 4. Because of the hierarchical naming features of *location* or *category*, significant amount of time may be saved by assigning severity levels to the parent of a plurality of children. The children will then automatically inherit the assigned severity as well.

```
static final Location loc =
Location.getLocation(com.sap.fooPackage);

loc.setEffectiveSeverity(Severity.INFO);
```

[00044] In one embodiment, by default, the source object (assuming the ascendant has not been assigned a severity level yet) is assigned *Severity.NONE.* As such, a developer may freely enable the output methods in the source code, but the actual logging is not activated until it is explicitly "switched on" when ready.

Specify Output Destination (503)

[00045] An output destination such as a log/trace file, a console, an output stream, etc, is assigned to each *Location* sub-controller or *Category* sub-controller. This can be accomplished via a method such as the following:

```
"loc.addLog(<log1>);."
```

[00046] There may be instances in which it is desirable to assign multiple output destinations to a single sub-controller. In such a case, a message generated from the sub-controller will be sent to all assigned output destinations simultaneously (e.g., to both a console and a file). This may be accomplished using the following: "`loc.addLog(new ConsoleLog())`"; and `loc.addLog(new FileLog("C:\\temp\\testOutput.log").`"

[00047] In one embodiment, as a default configuration, console output destinations 221 are assigned human-readable formatters 231 and file output destinations 220 are assigned list formatters 230. Of course, the default configurations are not required. For example, to switch to an XML Formatter for a file output, the following exemplary method may be called: "loc.addLog(new FileLog("C:\temp\testOutput.log", new XMLFormatter())." In an embodiment in which a file log already exists the following exemplary method may be called: " <filelog>.setFormatter(new XMLFormatter())."

Enable Output Messages (504)

[00048] After the source is defined, the severity levels are assigned and the destination is properly specified, output statements may be inserted in appropriate places in the program code. For the sake of explanation, the output methods may be logically

categorized into three groups: (1) typical message output with severity; (2) output denoting the flow of a program; and (3) master gate.

[00049] (1) *Typical Message Output With Severity*: Table 1 illustrates the format for a typical output message with severity, where the designated severity level is masked with "xxxx." The "T" designates a typical message output.

TABLE 1

Location	Category
xxxxT(String message)	xxxxT(Location loc, String message)
xxxxT(String subloc, String message)	xxxxT(Location loc, String subloc, String message)
xxxxT(String message, Object[] args)	xxxxT(Location loc, String message, Object[] args)
xxxxT(String subloc, String message, Object[] args)	xxxxT(Location loc, String subloc, String message, Object[] args)

[00050] A pattern exists in method overloading which evolves around the core argument: *message*. The addition of *subloc, args* offers the flexibility for developers to log messages using the level of detail that they need. Understanding these arguments helps when selecting the heavily overloaded methods.

[00051] One difference in the API between *Location* and *Category* is an additional *loc* argument in *Category* output methods. As described above, log messages are typically written with respect to a source code area. This proves to be very helpful for logging analysis. By specifying the *loc* argument, a programmer may indicate that the

message should be written as a trace message associated with the *loc* object. By properly configuring *loc*, logging can be just performed once and piped for both message types (e.g., logs & traces) simultaneously. This technique works for *Location* as well, and the API is available to specify the *category* argument. These techniques are explained in more detail below (section entitled "Relationship Between location and Category).

[00052] The *subloc* argument is treated as the method name of the source class that the message is generated from. This is optional, but with this argument included in the trace/log, the picture when doing analysis will be much clearer, for example, because different arguments can be specified for overloaded methods). The actual message to be output is placed in the argument *message*. A designation may be selected that meaningfully describes the situation/problem.

[00053] An array of additional arguments that are informative, e.g. dynamic information may be included in the message. In one embodiment, this is achieved by using `java.text.MessageFormat` API to resolve arguments.

[00054] Referring again to the code example set forth above, the following is an example working from the object *Location*:

```
package com.sap.fooPackage;
import com.sap.tc.logging.*;
public class Node {
  private static final Location loc =
Location.getLocation("com.sap.fooPackage.Node");
public void announce(Object o) {
    String method = "announce(java.lang.Object)";
```

```
        try {
        // do something...eg. connecting to DB, perform
    certain actions
          loc.debugT(method, "Connecting to ….");
          //minor error in writing something
          loc.warningT(method,
                       "Problems in row {0} to {1}",
                       new Object[] {row1, rowN});
          //finish successfully
          loc.infoT(method, "DB action done successfully");
    }
    catch (Exception e) {
    }
  }  // method announce
}  // class Node
```

[00055] Potential output is as follows, assuming the human-readable formatter is used (e.g., to display on a console):

```
May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main] Debug:
Connecting to ….
May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main] Warning:
Problems in row 15 to 18
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce
[main] Info: DB action done successfully
```

[00056] The following is another example, working from the object *Category:*

```
package com.sap.fooPackage;
import com.sap.tc.logging.*;
public class Node {
  private static final Location loc =
Location.getLocation("com.sap.fooPackage.Node");
  private static final Category cat =
```

```
Category.getCategory("/System/Database");

  public void store() {
    try {
     // Write object data to database ...
    }
    catch (FailedRegistrationException e) {
      cat.errorT(loc,
                 "store()",
     "Error storing node {0} in database.",
                 new Object[] {this});
    }
  } // method store
} // class Node
```

[00057] Note that the output will be identical to the previous example, assuming the default setting is used (e.g., using a human-readable formatter).

[00058] (2) *Output Denoting the Flow of a Program:* This feature is typically only used for *Location* sub-controllers. Tracing the flow of a program may be comprised of several components, for example: (a) entering, (b) exiting, (c) throwing, and (d) assertion.

[00059] (a) Entering: Outputs a default message (e.g., "Entering Method" with *Path* severity) indicating that it is entering a source block, as shown by Table 2:

TABLE 2

Method	Description
Entering()	Entering a source block in general

Entering(String subloc)	Specify the method name in *subloc*
Entering(Object[] args)	A general source block with arguments: "Entering method with <args>"
Entering(String subloc, Object[] args)	Same as above but with specific method name

[00060] (b) Exiting: Output is a default message (e.g., "Exiting Method" with Path severity) indicating that it is leaving a source block, as shown by Table 3:

TABLE 3

Method	***Description***
Exiting()	Exiting a source block in general. As long as the methodname (subloc) is specified in '*entering*', it is not necessary to provide *subloc* as argument here anymore. See the result of the following sample code.
Exiting(String subloc) //DEPRECATED	Specify the method name in *subloc*
Exiting(Object res)	A general source block with result: "Exiting method with <res>"
Exiting(String subloc, Object res)	Same as above but with specific method name

[00061] To reiterate, refer to the sample code with method *announce(Object o)*:

```
public void announce(Object o) {
  String method = "announce(java.lang.Object)";
  loc.entering(method);
  try {
  }
  catch (Exception e) {
  }
  loc.exiting();
}
```

[00062] Potential output, assuming the simplest case with ConsoleLog and default TraceFormatter:

```
May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main] Path: Entering
method
     May 3, 2001 6:54:18 PM
com.sap.fooPackage.Node.announce [main] Path: Exiting
method
```

[00063] (c) Throwing: Warning message ("Throwing ..."), indicating that the source block is about to throw an exception, as shown by Table 4:

TABLE 4

Method	***Description***
throwing(Throwable exc)	About to throw the exception *exc*
Throwing(String subloc, Throwable exc)	Same as above but with specific method name

[00064] (d) Assertion: used to test a condition and output an error message, normally with the assertion included (e.g., "Assertion failed: <assertion test>") when the evaluation is false, as shown by Table 5:

TABLE 5

Method	*Description*
Assertion(Boolean assertion, String desc)	Evaluate the assertion, if false, print *desc* with the default message: "Assertion failed: <*desc*>" where <*desc*> is the assertion test itself, e.g. 5 > 3
Assertion(String subloc, Boolean assertion, String desc)	Same as above but with specific method name

[00065] To reiterate, refer to the sample code with method *announce(Object o)*:

```
public void announce(Object o) {
String method = "announce(java.lang.Object)";
loc.assertion(method, 5<3,  "Stupid comparison");
try {
            }
catch (Exception e) {
    loc.throwing(method, e);
          }
}
```

[00066] The following is the potential output, again assuming a human-readable log formatter:

May 3, 2001 6:54:18 PM

com.sap.fooPackage.Node.announce [main]

Error: Assertion failed: Stupid comparison

May 3, 2001 6:54:18 PM

com.sap.fooPackage.Node.announce [main]

Warning: Throwing java.io.FileNotFoundException:

C:\Not_Exist\zzzzz.log (The system cannot find the path specified)

[00067] *(3)* Master Gate: In one embodiment, all of the other output methods (with severity) are ultimately routed through a method, referred to herein as LogT, to actually perform any logging/tracing.

Location: These are similar to the first type of method, *xxxxT( )*, but only with an additional severity argument at the beginning:

```
logT(int severity, String message)
logT(int severity, String subloc, String message)
logT(int severity, String message, Object[] args)
     logT(int severity, String subloc, String message,
     Object[] args)
```

Category: (Similar scenario to *Location):*

```
logT(int severity, Location loc, String message)
logT(int severity, Location loc, String subloc, String
message)
logT(int severity, Location loc, String message,
Object[] args)
logT(int severity, Location loc, String subloc, String
message,
Object[] args)
```

GROUPING RELATED LOG/TRACE MESSAGES

[00068] Often, it is useful to put several related messages together into one context. A typical example would be all trace messages stemming from one method call. In case of a database log, another example would be the messages representing the

different database operations together forming one logical transaction. A formatter or log viewer can utilize this context information to visualize relations using, for example, indentation or tree controls. Groups are one mechanism to express such context information.

[00069] In one embodiment, a group is established via a call to `openGroup`. This call is based on the same conditions as output calls, that is, the group is opened depending on a severity and optional categories. After generating output the group is ended via a call to `closeGroup`. Note that there should be a balancing call of `closeGroup` for any call of `openGroup`. Even if an `openGroup` call did not open the group, `closeGroup` is matched with the call to `openGroup`. In case of success, in between the two calls, all output calls are assigned to the group. Messages may even be generated with the same condition as the group via the `groupT` and `group` output calls. These calls are also used to emit the opening and closing messages of a group, which are the first such messages emitted after the `openGroup` and `closeGroup` calls, respectively.

[00070] In the above method, for example, the following piece of code could be written. The message right after the `openGroup` call is the opening message of the group, the message after `closeGroup` is its closing message. Note that the `groupT` calls do not need a severity, a location or a method name, as these are fetched from the active group.

```
cat.openGroup(Severity.INFO,
              loc,
              method);
cat.groupT("Start storing tree with root {0}.",
           new Object[] {this});
...
```

```
cat.groupT("Storing subnode {0}.",
           new Object[] {node});
...
cat.closeGroup();
cat.groupT("Finished storing tree.");
```

ESTABLISHING RELATIONSHIPS BETWEEN CATEGORY & LOCATION

[00071] It is a common practice to look at log messages and trace messages together when performing a diagnosis. A correlation between a problematic logical area and the source code location that generates the problem is highly desirable. For example, if an error occurs when closing the database, the actual location of the source code (from which class, which method, with what argument(s)) is reported as well.

[00072] An embodiment of the invention simplifies the generation of both log and trace messages in parallel, with the use of *category* and *location*. This will be illustrated with the following example. For simplicity, the example only shows the attachment of a *location* to a *category* (as used herein, a *location* associated with the category is referred to as a "relative" of the category). However, the same general principles apply to the attachment of a *category* to a *location*.

[00073] More than one *category* may be associated with a *location* at each logging. However, in one embodiment, only one *location* is assigned for one *category*. Refer to the output method APIs of each class:

```
Package com.sap.fooPackage;
import com.sap.tc.logging.*;
public class Node {
```

```
    private static final Location loc =
  Location.getLocation("com.sap.fooPackage.Node");
    private static final Category objMgmt =
  Category.getCategory("/Objects/Management");
    public void announce(Object o) {
      final String method =
  "announce(java.lang.Object)";
      loc.entering(method, new Object[] {o});
      try {
        // Register object ...
      }
      catch (RegistrationException e) {
        objMgmt.errorT(loc,
                       method,
                     "Error registering object {0}.",
                       new Object[] {o});
      }
      loc.exiting();
    }  // method announce
}  // class Node
```

[00074] In order to output all the trace and log messages highlighted in the example above, the following severity setting may be employed:

```
loc.setEffectiveSeverity(Severity.PATH);
objMgmt.setEffectiveSeverity(Severity.ERROR);
conLog = new ConsoleLog();
```

```
loc.addLog(conLog);
objMgmt.addLog(conLog);
```

[00075] With respect to the output line from the *category* `'objMgmt'`, it will output two messages simultaneously: one log message and one trace message. They will have the same message id for cross-referencing each other. This greatly simplifies the analysis.

[00076] If the *location* has stricter severity setting (e.g. default *Severity.NONE*) all trace output may be suppressed, including the one from the *category*. In other words, that output line will NOT produce two messages simultaneously, but only the log message.

[00077] More advanced configuration may be employed regarding the correlated *category* and *location* source objects. For example, consider *Category* "/Objects/Management" where the only focus may be some extreme situations, that is, messages with severity *FATAL*. Several source code *locations* (`'com.sapmarkets.xxx.a'`, `'com.sapmarkets.xxx.b'`, ...) can result in a fatal condition in this logical area, and for certain reasons, one of them may be of particular interest (e.g. `'com.sapmarkets.xxx.a'`). As such, it would beneficial to have the ability to generate additional output messages, including all with severity *INFO* or above, related with this *location* only, while maintaining *FATAL* for the rest.

CONFIGURING A FORMATTER

[00078] Referring again to FIG. 2, formatters (e.g., formatters 230, 231, and 232) may be used to format a trace/log message. In an embodiment, the formatter (e.g.,

formatter 231) may be configured to provide a desired message format. For example, in an embodiment, the log/trace message format provided by a formatter is defined by one or more properties in a configuration file. In such an embodiment, the message format may be configured by providing/removing/altering values for the properties defined in the configuration file. Since the values may be provided/removed/altered during runtime, a formatter may be configured without recompiling any source code.

[00079] FIG. 7a illustrates selected elements of an exemplary configuration file 700, according to an embodiment of the invention. In the illustrated embodiment, properties are defined with one or more key-value-pairs. Key-value-pairs may have the following format:

<location>.attribute = value; and

<category>.attribute = value.

[00080] For example, configuration file 700 includes key-value-pairs 710 and 720. Key-value-pair 710 defines a severity level for location 712 (e.g., com.sapmarkets.usermanagement) by setting attribute 714 equal to value 716 (e.g., WARNING). Key-value-pair 720, defines an output destination for location 712 by setting attribute 722 to output destination 724. In an alternative embodiment of the invention, configuration file 700 uses a format different than the key-value-pair format. In an embodiment, a particular type of formatter is, by default, assigned to a given output destination. For example, a trace formatter may be assigned to a console and list formatter may be assigned to a file log, by default. As is further described below, with

reference to FIG. 8, the default formatter may be replaced with a different formatter by setting a value in, for example, configuration file 700.

[00081] FIG. 7b illustrates a default log/trace message format as defined by configuration file 700. In an embodiment, a formatter initially provides log/trace messages according to the default message format. As is further described below, with reference to FIG. 8, the default message format may be replaced by defining a particular message format in a configuration file. In one embodiment, the default message format is defined by a string of placeholders (and associated values) such as:

"%24d %-40l [%t] %s: %m."

Each placeholder (and associated value) may define a field in the message format. In such an embodiment, the string shown above may correspond to an output such as:

"Jan 01, 2001 10:10:00 PM com.sapmarkets.FooClass.fooMethod [main] Fatal: A sample fatal message."

Table 6 provides a description of selected placeholders according to an embodiment of the invention.

TABLE 6

Placeholder	Description
%d	timestamp in readable form
%l	the location of origin (eg: com.sapmarkets.FooClass.fooMethod)

%c	the log controller the message was issued through (eg: com.sapmarkets.FooClass)
%t	the thread that emitted the message
%s	the message severity
%m	the formatted message text
%I	the message id
%p	the time stamp in milliseconds since January 1, 1970 00:00:00 GMT
%g	the group identification

[00082] In an embodiment, a configuration file may be used to, for example, change the default formatter assignment and/or the default message format. FIG. 8 illustrates selected elements of an exemplary configuration file 800, according to an embodiment of the invention in which the message format is configurable. In the embodiment illustrated in FIG. 8, the message format may be configured without recompiling any source code because the integrated logging/tracing system accesses configuration file 800 to determine the value of defined properties such as the format of a log/trace message.

[00083] Key-value-pair 805 assigns severity level 815 to source 810. In addition, key-value-pair 820 assigns two output destination destinations to source 810: console 825 and log file 830. Formatter 840 is assigned to console 825 by key-value-pair 835. In an

embodiment, an identifier may be assigned to a formatter to uniquely (within the configuration file) identify the formatter. In the illustrated embodiment, key-value-pair 835 also provides identifier 845 (e.g., TraceNoDate) to identify formatter 840.

[00084] In an embodiment, defining a desired message format is accomplished by setting an attribute (e.g., a pattern attribute) for a formatter in configuration file 800. Key-value-pair 850 illustrates setting pattern attribute 855 to value 860 for formatter 840. In the illustrated embodiment, value 860 is a string of placeholders and associated values (e.g., placeholders/values 861-864) that define, for example, a log/trace message having four fields. As identifier 845 (e.g., TraceNoDate) suggests, value 860 defines a message format that does not include a timestamp.

[00085] In an embodiment, value 860 may (or may not) include any of the fields defined by the placeholders (and related values) specified in Table 6. Similarly, in an embodiment additional and/or different fields may be defined by additional and/or different placeholders (and associated values). In an alternative embodiment of the invention, the fields of a trace/log message may be specified by a value that has format different than the placeholder format shown in FIGs. 7-8.

[00086] Turning now to FIG. 9, the particular methods associated with embodiments of the invention are described in terms of computer software and hardware with reference to a flowchart. The methods to be performed by a unified logging service with a logging formatter may constitute state machines or computer programs made up of computer-executable instructions. Describing the methods by reference to a flowchart enables one of ordinary skill in the art to develop such programs including such

instructions to carry out the methods on suitably configured computing devices (e.g., one or more processors of a node) executing the instructions from computer-accessible media. The computer-executable instructions may be written in a computer programming language or may be embodied in firmware logic. If written in a programming language conforming to a recognized standard, such instructions can be executed on a variety of hardware platforms and for interface to a variety of operating systems. In addition, embodiments of the invention are not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein. Furthermore, it is common in the art to speak of software, in one form or another (e.g., program, procedure, process, application, etc.), as taking an action or causing a result. Such expressions are merely a shorthand way of saying that execution of the software by a computing device causes the device to perform an action or produce a result.

[00087] FIG. 9 is a flow diagram illustrating certain aspects of a method for configuring the message format of a log/trace message, according to an embodiment of the invention. Referring to process block 910, an instance of a tracing controller associated with specified program code regions of an application is created. In an embodiment, the tracing controller instance receives and processes tracing method calls generated by the application when the specified program code regions are executed. In an embodiment, a threshold severity level for tracing method calls may be defined by a value in a configuration file (e.g., configuration file 800, shown in FIG. 8). Tracing method calls having a severity level below this threshold may not be processed.

[00088] Referring to process block 920, an instance of a logging controller associated with specified categories related to a network is created. In an embodiment, the logging controller instance receives and processes logging method calls from network components associated with the categories. In an embodiment, a threshold severity level for logging method calls may be defined by a value in a configuration file (e.g., configuration file 800, shown in FIG. 8). Logging method calls having a severity level below this threshold may not be processed.

[00089] Referring to process block 930, an output destination to receive log/trace messages is specified. In an embodiment, the output destination may receive log/trace messages from the tracing controller instance and/or the logging controller instance. In an embodiment, the output destination for a tracing controller instance and/or a logging controller instance may be defined by a value in a configuration file. In such an embodiment, specifying the output destination broadly refers to providing/setting/changing a value representing the output destination in the configuration file. In one embodiment, the possible output destinations include a console and/or a file log.

[00090] Referring to process block 940, a formatter to provide a message format for log/trace messages is selected. In an embodiment, one or more distinct formatters may be selected for each source (e.g., for each logging controller instance and/or each tracing controller instance). The relationship between a source and a formatter may be defined in a configuration file (e.g., configuration file 800, shown in FIG. 8). In such an embodiment, selecting the formatter broadly refers to providing/setting/changing a value representing the formatter in the configuration file. In one embodiment, the available

formatter types include: a list formatter, a human-readable formatter, and/or a markup language formatter.

[00091] Referring to process block 950, the selected formatter is configured, for example, without recompiling any source code. In an embodiment, the message format provided by the selected formatter is defined by one or more properties in a configuration file. For example, the message format may be defined to have one or more fields that are specified by a string of placeholders (and associated values). In such an embodiment, configuring the formatter may include configuring the message format by providing/setting/changing the values of the properties defined in the configuration file. In one embodiment, configuring the formatter may also include providing/setting/changing an identifier to uniquely identify (within the configuration file) the formatter.

[00092] FIG. 10 is a block diagram of computing device 1000 implemented according to an embodiment of the invention. Computing device 1000 may include: one or more processors 1010, memory 1020, one or more Input/Output (I/O) interfaces 1030, network interface(s) 1040, configuration file 1050, and integrated logging/tracing system 1060. The illustrated elements may be connected together through system interconnect 1070. One or more processors 1010 may include a microprocessor, microcontroller, field programmable gate array (FPGA), application specific integrated circuit (ASIC), central processing unit (CPU), programmable logic device (PLD), and similar devices that access instructions from system storage (e.g., memory 1020), decode them, and execute those instructions by performing arithmetic and logical operations.

[00093] Integrated logging/tracing system 1060 enables computing device 1000 to provide an integrated logging and tracing architecture. Integrated logging/tracing system 1060 may be executable content, control logic (e.g., ASIC, PLD, FPGA, etc.), firmware, or some combination thereof, in an embodiment of the invention. In embodiments of the invention in which integrated logging/tracing system 1060 is executable content, it may be stored in memory 1020 and executed by processor(s) 1010.

[00094] Configuration file 1050 defines one or more properties for one or more configurable attributes of integrated logging/tracing system 1060. Configuration file 1050 may be executable content, control logic (e.g., ASIC, PLD, FPGA, etc.), firmware, or some combination thereof, in an embodiment of the invention. In embodiments of the invention in which configuration file 1050 is executable content, it may be stored in memory 1020 and manipulated by processor(s) 1010.

[00095] Memory 1020 may encompass a wide variety of memory devices including read-only memory (ROM), erasable programmable read-only memory (EPROM), electrically erasable programmable read-only memory (EEPROM), random access memory (RAM), non-volatile random access memory (NVRAM), cache memory, flash memory, and other memory devices. Memory 1020 may also include one or more hard disks, floppy disks, ZIP disks, compact disks (e.g., CD-ROM), digital versatile/video disks (DVD), magnetic random access memory (MRAM) devices, and other system-readable media that store instructions and/or data. Memory 1020 may store program modules such as routines, programs, objects, images, data structures, program data, and other program modules that perform particular tasks or implement particular abstract data types that facilitate system use.

[00096] One or more I/O interfaces 1030 may include a hard disk drive interface, a magnetic disk drive interface, an optical drive interface, a parallel port, serial controller or super I/O controller, serial port, universal serial bus (USB) port, a display device interface (e.g., video adapter), a network interface card (NIC), a sound card, modem, and the like. System interconnect 1070 permits communication between the various elements of computing device 1000. System interconnect 1070 may include a wide variety of signal lines including one or more of a memory bus, peripheral bus, local bus, host bus, bridge, optical, electrical, acoustical, and other propagated signal lines.

[00097] It should be appreciated that reference throughout this specification to "one embodiment" or "an embodiment" means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Therefore, it is emphasized and should be appreciated that two or more references to "an embodiment" or "one embodiment" or "an alternative embodiment" in various portions of this specification are not necessarily all referring to the same embodiment. Furthermore, the particular features, structures or characteristics may be combined as suitable in one or more embodiments of the invention.

[00098] Similarly, it should be appreciated that in the foregoing description of exemplary embodiments of the invention, various features of the invention are sometimes grouped together in a single embodiment, figure, or description thereof for the purpose of streamlining the disclosure aiding in the understanding of one or more of the various inventive aspects. This method of disclosure, however, is not to be interpreted as reflecting an intention that the claimed invention requires more features than are

expressly recited in each claim. Rather, as the following claims reflect, inventive aspects lie in less than all features of a single foregoing disclosed embodiment. Thus, the claims following the detailed description are hereby expressly incorporated into this detailed description, with each claim standing on its own as a separate embodiment of this invention.
